32 OpenMP Traps For C++ Developers

Alexey Kolosov
OOO "Program Verification Systems"
Evgeniy Ryzhkov
OOO "Program Verification Systems"
Andrey Karpov
OOO "Program Verification Systems"

May 2008

Abstract

Since multi-core systems are spreading fast, the problem of parallel programming becomes more and

more urgent. However, even the majority of experienced developers are new to this sphere. The existing compilers and code analyzers allow finding some bugs, which appear during parallel code development. However, many errors are not diagnosed. The article contains description of a number of errors, which lead to incorrect behavior of parallel programs created with OpenMP.

Introduction

Parallel programming appeared a long time ago. The first multiprocessor computer was created in 1960s. However, processors' performance increase has been achieved through clock frequency increment and multiprocessor systems have been rare until recently. The clock frequency increment slows down nowadays and processors' performance increase is achieved through multiple cores. Multi-core processors are spread widely, therefore the problem of parallel programming becomes more and more urgent. Earlier it was enough to install a CPU with a higher clock frequency or larger cash memory to increase a program's performance. Nowadays this approach is useless and a developer will have to modify the program in order to increase the program's performance.

Since parallel programming begins gaining in popularity only now, the process of an existing application parallelization or a new parallel program creation may become very problematic even for experienced developers since this sphere is new for them. Currently existing compilers and code analyzers allow finding only some (very few) potential errors. All other errors remain unrecorded and may increase debug and testing time significantly. Besides that, almost all errors of this kind cannot be stably reproduced. The article concerns the C++ language, since C++ programs are usually demanded to work fast. Since Visual Studio 2005 & 2008 support the OpenMP 2.0 standard, we will concern the OpenMP technology. OpenMP allows you to parallelize your code with minimal efforts - all you need to do is to enable the /openmp compiler option and add the needed compiler directives describing how the program's execution flow should be parallelized to your code.

This article describes only some of the potential errors which are not diagnosed by compilers, static code analyzers and dynamic code analyzers. However, we hope that this paper will help you understand some peculiarities of parallel development and avoid multiple errors.

Also, please note that this paper contains research results, which will be used in the VivaMP static analyzer development (http://www.viva64.com/vivamp.php). The static analyzer will be designed to find errors in parallel programs created with OpenMP. We are very interested in receiving feedback on this article and learning more patterns of parallel programming errors.

The errors described in this article are split into logical errors and performance errors similar to the approach used in one of the references [1]. Logical errors are errors, which cause unexpected results, i.e. incorrect program behavior. Performance errors are errors, which decrease a program's performance.

First of all, let us define some specific terms which will be used in this article:

Directives are OpenMP directives which define code parallelization means. All OpenMP directives have the appearance of #pragma omp ...

Clauses are auxiliary parts of OpenMP directives. Clauses define how a work is shared between threads, the number of threads, variables access mode, etc.

Parallel section is a code fragment to which the #pragma omp parallel directive is applied.

The article is for developers who are familiar to OpenMP and use the technology in their programs. If you are not familiar with OpenMP, we recommend that you take a look at the document [2]. A more detailed description of OpenMP directives, clauses, functions and environment variables can be found in the

OpenMP 2.0 specification [3]. The specification is duplicated in the MSDN Library and this form of specification is more handy, then the one in the PDF format.

Now, let us describe the potential errors which are badly diagnosed by standard compilers or are not diagnosed at all.

Logical errors

1. Missing /openmp option

Let's start with the simplest error: OpenMP directives will be ignored if OpenMP support is not enabled in the compiler settings. The compiler will not report an error or even a warning, the code simply will not work the way the developer expects.

OpenMP support can be enabled in the "Configuration Properties | C/C++ | Language" section of the project properties dialog.

2. Missing parallel keyword

OpenMP directives have rather complex format, therefore first of all we are considering the simplest errors caused by incorrect directive format. The listings below show incorrect and correct versions of the same code:

Incorrectly:

```
#pragma omp for
... //your code
```

Correctly:

```
#pragma omp parallel for
... // your code
```

```
#pragma omp parallel
{
  #pragma omp for
  ... //your code
}
```

The first code fragment will be successfully compiled, and the #pragma omp for directive will be simply ignored by the compiler. Therefore, a single thread only will execute the loop, and it will be rather difficult for a developer to find this out. Besides the #pragma omp parallel for directive, the error may also occur with the #pragma omp parallel sections directive.

3. Missing omp keyword

A problem similar to the previous one occurs if you omit the omp keyword in an OpenMP directive [4]. Let's take a look at the following simple example:

Incorrectly:

```
#pragma omp parallel num_threads(2)
```

```
{
   #pragma single
   {
     printf("me\n");
   }
}
```

Correctly:

```
#pragma omp parallel num_threads(2)
{
   #pragma omp single
   {
     printf("me\n");
   }
}
```

The "me" string will be printed twice, not once. The compiler will report the "warning C4068: unknown pragma" warning. However, warnings can be disabled in the project's properties or simply ignored by a developer.

4. Missing for keyword

The #pragma omp parallel directive may be applied to a single code line as well as to a code fragment. This fact may cause unexpected behavior of the for loop shown below:

```
#pragma omp parallel num_threads(2)
for (int i = 0; i < 10; i++)
   myFunc();
```

If the developer wanted to share the loop between two threads, he should have used the #pragma omp parallel for directive. In this case the loop would have been executed 10 times indeed. However, the code above will be executed once in every thread. As the result, the myFunc function will be called 20 times. The correct version of the code is provided below:

```
#pragma omp parallel for num_threads(2)
for (int i = 0; i < 10; i++)
   myFunc();
```

5. Unnecessary parallelization

Applying the #pragma omp parallel directive to a large code fragment may cause unexpected behavior in cases similar to the one below:

```
#pragma omp parallel num_threads(2)
{
    ... // N code lines
    #pragma omp parallel for
    for (int i = 0; i < 10; i++)
    {
         myFunc();
    }
}
```

In the code above a forgetful or an inexperienced developer who wanted to share the loop execution between two threads placed the parallel keyword inside a parallel section. The result of the code execution will be similar to the previous example: the myFunc function will be called 20 times, not 10. The correct version of the code should look like this:

```
#pragma omp parallel num_threads(2)
{
    ... // N code lines
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
         myFunc();
    }
}
```

6. Incorrect usage of the ordered clause

The ordered directive may cause problems for developers who are new to OpenMP [1]. Let's consider the following sample:

Incorrectly:

```
#pragma omp parallel for ordered
for (int i = 0; i < 10; i++)
{
    myFunc(i);
}
```

Correctly:

```
#pragma omp parallel for ordered
for (int i = 0; i < 10; i++)
{
    #pragma omp ordered
    {
           myFunc(i);
    }
}
```

In the first code fragment the ordered clause will be simply ignored, because its scope was not specified. The loop will still be executed in a random order (which may sometimes become ascending order if you're lucky).

7. Redefining the number of threads in a parallel section

Now, let us consider more complex errors, which may be caused by insufficient understanding of the OpenMP standard. According to the OpenMP 2.0 specification [3] the number of threads cannot be redefined inside a parallel section. Such an attempt will cause run-time errors and program termination of a C++ program. For example:

Incorrectly:

```
#pragma omp parallel
{
    omp_set_num_threads(2);
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
         myFunc();
    }
}
```

Correctly:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
      myFunc();
    }
}
```

Correctly:

```
omp_set_num_threads(2)
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
         myFunc();
    }
}
```

8. Using a lock variable without initializing the variable

According to the OpenMP 2.0 specification [3] all lock variables must be initialized via the omp_init_lock or omp_init_nest_lock function call (depending on the variable type). A lock variable can be used only after initialization. An attempt to use (set, unset, test) an uninitialized lock variable In a C++ program will cause a run-time error.

Incorrectly:

```
omp_lock_t myLock;
#pragma omp parallel num_threads(2)
{
    ...
    omp_set_lock(&myLock);
    ...
}
```

Correctly:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel num_threads(2)
{
    ...
    omp_set_lock(&myLock);
    ...
}
```

9. Unsetting a lock from another thread

If a lock is set in a thread, an attempt to unset this lock in another thread will cause unpredictable behavior [3]. Let's consider the following example:

Incorrectly:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
    }
    #pragma omp section
    {
          ...
          omp_unset_lock(&myLock);
          ...
    }
}
```

This code will cause a run-time error in a C++ program. Since lock set and unset operations are similar to entering and leaving a critical section, every thread, which uses locks should perform both operations. Here is a correct version of the code:

Correctly:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
          omp_unset_lock(&myLock);
          ...
    }
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
          omp_unset_lock(&myLock);
          ...
    }
}
```

10. Using a lock as a barrier

The omp_set_lock function blocks execution of a thread until the lock variable becomes available, i.e. until the same thread calls the omp_unset_lock function. Therefore, as it has already been mentioned in the description of the previous error, each of the threads should call both functions. A developer with insufficient understanding of OpenMP may try to use the omp_set_lock function as a barrier, i.e. instead of the #pragma omp barrier directive (since the directive cannot be used inside a parallel section, to which the #pragma omp sections directive is applied). As the result the following code will be created:

Incorrectly:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
    }
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          omp_unset_lock(&myLock);
          ...
    }
}
```

Sometimes the program will be executed successfully. Sometimes it will not. This depends on the thread which finishes its execution first. If the thread which blocks the lock variable without releasing it will be finished first, the program will work as expected. In all other cases the program will infinitely wait for the thread, which works with the lock variable incorrectly, to unset the variable. A similar problem will occur if the developer will place the omp_test_lock function call inside a loop (and that is the way the function is usually used). In this case the loop will make the program hang, because the lock will never be unset.

Since this error is similar to the previous one, the fixed version of the code will remain the same:

Correctly:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
          omp_unset_lock(&myLock);
          ...
    }
    #pragma omp section
    {
          ...
          omp_set_lock(&myLock);
          ...
          omp_unset_lock(&myLock);
          ...
    }
}
```

11. Threads number dependency

The number of parallel threads created during a program execution is not a constant value in general case. The number is usually equal to the number of processors by default. However, a developer can specify the number of threads explicitly (for example, using the omp_set_num_threads function or the num_threads clause, which has higher priority than the function). The number of threads can also be specified via the OMP_NUM_THREADS environment variable, which has the lowest priority. Therefore, the number of threads, which currently execute a parallel section, is a very unreliable value. Besides, the value may vary from one machine to another. The behavior of your code should not depend on the number of threads, which execute the code, unless you are entirely sure that this is really necessary.

Let's consider an example from the article [5]. The following program should have printed all letters of the English alphabet according to the developer's plan.

Incorrectly:

```
omp_set_num_threads(4);
#pragma omp parallel private(i)
{
    int LettersPerThread = 26 / omp_get_num_threads();
    int ThisThreadNum = omp_get_thread_num();
    int StartLetter = 'a' + ThisThreadNum * LettersPerThread;
    int EndLetter = 'a' + ThisThreadNum * LettersPerThread + LettersPerThread;
    for (int i=StartLetter; i<EndLetter; i++)
         printf ("%c", i);
}
```

However, only 24 of 26 letters will be printed. The cause of the problem is that 26 (the total number of letters) do not contain 4 (the number of threads). Therefore, the two letters remaining will not be printed. To fix the problem one can either significantly modify the code so that the code will not use the number of threads or share the work between a correct number of threads (e.g. 2 threads). Suppose the developer decided not to use the number of threads in his program and let the compiler share work between threads. In this case the fixed version of the code will be similar to the following one:

Correctly:

```
omp_set_num_threads(4);
#pragma omp parallel for
for (int i = 'a'; i <= 'z'; i++)
{
     printf ("%c", i);
}
```

All iterations of the loop will surely be executed. One can specify the way the iterations are shared between threads using the schedule clause. Now, the compiler will share work between the threads and he will never forget about the two "additional" iterations. In addition, the resulting code is significantly shorter and readable.

12. Incorrect usage of dynamic threads creation

The dynamic keyword may appear in two different contexts in OpenMP: in the schedule(dynamic) clause and in the OMP_DYNAMIC environment variable, which makes a little mess of this. It is important to understand the difference between the two cases. One should not think that the schedule(dynamic) clause can be used only if the OMP_DYNAMIC variable is equal to true. The two cases are not related at all, actually.

The schedule(dynamic) clause means that iterations of a loop are split into chunks, which are dynamically shared between threads. When a thread finishes execution of a chunk, the thread will start executing the following "portion". If we apply this clause to the previous example, each of the 4 threads will print 6 letters and then the thread, which will become free first, will print the last 2 letters.

The OMP_DYNAMIC variable sets, whether the compiler can define the number of threads dynamically. The cause of a possible problem with this variable is that the variable's priority is even higher than the one of the num_threads clause. Therefore, if the variable's value is equal to true, the setting overrides num_threads, omp_set_num_threads and OMP_NUM_THREADS. If a program's behavior depends on the number of threads, this may cause unexpected results. This is another argument for creating code, which does not depend on the number of threads.

As experience has shown, the value of the OMP_DYNAMIC environment variable is equal to false by default in Visual Studio 2008. However, there is no guarantee that this situation will remain unchanged in the future. The OpenMP specification [3] states that the variable's value is implementation-specific. Therefore, if the developer from the previous example chose an easier way and decided to use the number of threads in his calculations instead of modifying the code significantly, he should make sure that the number of threads would always be equal to the one he needs. Otherwise the code will not work correctly on a four-processor machine.

Correctly:

```
if (omp_get_dynamic())
  omp_set_dynamic(0);
omp_set_num_threads(2);
#pragma omp parallel private(i)
{
    int LettersPerThread = 26 / omp_get_num_threads();
    int ThisThreadNum = omp_get_thread_num();
    int StartLetter = 'a' + ThisThreadNum * LettersPerThread;
    int EndLetter = 'a' + ThisThreadNum * LettersPerThread + LettersPerThread;
    for (i=StartLetter; i<EndLetter; i++)
          printf ("%c", i);
}
```

13. Concurrent usage of a shared resource

If we modify the previous example's code so that the code prints at least two or more letters at a time (not one by one in a random order as it currently does), we will observe one more parallel programming problem, the problem of concurrent shared resource usage. In this case the resource is the application's console. Let's consider a slightly modified example from the article [6].

Incorrectly:

```
#pragma omp parallel num_threads(2)
{
    printf("Hello World\n");
}
```

In spite of the developer's expectations, the program's output on a two-processor machine will be similar to the following two lines:

```
HellHell oo WorWlodrl
d
```

The behavior is caused by the fact that the string output operation is not atomic. Therefore, the two threads will print their characters simultaneously. The same problem will occur if you use the standard output thread (cout) or any other object accessible to the threads as a shared variable.

If it is necessary to perform an action, which changes a shared object's state, from two threads, one should make sure that the action is performed by a single thread at a time. One can use locks or critical sections to achieve this. The most preferable approach will be discussed further.

Correctly:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp critical
    {
          printf("Hello World\n");
    }
}
```

14. Shared memory access unprotected

This error is described in the article [1]. The error is similar to the previous one: if several threads are modifying a variable's value concurrently, the result is unpredictable. However, the error is considered separately from the previous one, because in this case the solution will be slightly different. Since an operation on a variable can be atomic, it is more preferable to use the atomic directive in this case. This approach will provide better performance than critical sections. Detailed recommendations on shared memory protection will be provided further.

Incorrectly:

```
int a = 0;
#pragma omp parallel
{
    a++;
}
```

Correctly:

```
int a = 0;
#pragma omp parallel
{
    #pragma omp atomic
    a++;
}
```

Another possible solution is to use the reduction clause. In this case every thread will get its own copy of the a variable, perform all the needed actions on this copy and then perform the specified operation to merge all the copies.

Correctly:

```
int a = 0;
#pragma omp parallel reduction(+:a)
{
    a++;
}
printf("a=%d\n", a);
```

The code above, being executed by two threads, will print the "a=2" string.

15. Using the flush directive with a reference type

The flush directive makes all the threads refresh values of shared variables. For example, if a thread assigns 1 to a shared variable a, it does not guarantee that another thread reading the variable will get 1. Please note that the directive refreshes only the variables' values. If an application's code contains a shared reference pointing to an object, the flush directive will refresh only the value of the reference (a memory address), but not the object's state. In addition, the OpenMP specification [3] states explicitly that the flush directive's argument cannot be a reference.

Incorrectly:

```
MyClass* mc = new MyClass();
#pragma omp parallel sections
{
    #pragma omp section
    {
          #pragma omp flush(mc)
          mc->myFunc();
          #pragma omp flush(mc)
    }
    #pragma omp section
    {
          #pragma omp flush(mc)
          mc->myFunc();
          #pragma omp flush(mc)
    }
}
```

The code below actually contains two errors: concurrent access to a shared object, which has already been described above, and usage of the flush directive with a reference type. Therefore, if the myFunc method changes the object's state, the result of the code execution is unpredictable. To avoid the errors one should get rid of concurrent usage of the shared object. Please note that the flush directive is executed implicitly at entry to and at exit from critical sections (this fact will be discussed later).

Correctly:

```
MyClass* mc = new MyClass();
#pragma omp parallel sections
{
    #pragma omp section
    {
          #pragma omp critical
          {
                 mc->myFunc();
          }
    }
    #pragma omp section
    {
          #pragma omp critical
          {
                mc->myFunc();
          }
    }
}
```

16. Missing flush directive

According to the OpenMP specification [3], the directive is implied in many cases. The full list of such cases will be provided further. A developer may count upon this fact and forget to place the directive in a place where it is really necessary. The flush directive is **not** implied in the following cases:

- At entry to for.
- At entry to or exit from master.
- At entry to sections.
- At entry to single.
- At exit from for, single or sections, if the nowait clause is applied to the directive. The clause removes implicit flush along with the implicit barrier.

Incorrectly:

```
int i = 0;
#pragma omp parallel num_threads(2)
{
    a++;
    #pragma omp single
    {
          cout << a << endl;
    }
}
```

Correctly:

```
int i = 0;
#pragma omp parallel num_threads(2)
{
    a++;
    #pragma omp single
    {
          #pragma omp flush(a)
          cout << a << endl;
    }
}
```

The latest version of the code uses the flush directive, but it is not ideal too. This version lacks of synchronization.

17. Missing synchronization

Besides the necessity of the flush directive usage, a developer should also keep in mind threads synchronization.

The corrected version of the previous example does not guarantee that the "2" string will be printed to the application's console window. The thread executing the section will print the value of the a variable which was actual at the moment the output operation was performed. However, there is no guarantee that both threads will reach the single directive simultaneously. In general case the value might be equal to "1" as well as "2". This behavior is caused by missing threads synchronization. The single directive means that the corresponding section should be executed only by a single thread. However, it is equiprobable that the section will be executed by the thread which finishes its execution first. In this case the "1" string will be printed. A similar error is described in the article [7].

Implicit synchronization via an implied barrier directive is performed only at exit from the for, single or sections directive, if the nowait clause is not applied to the directive (the clause removes the implicit barrier). In all other cases the developer should take care of the synchronization.

Correctly:

```
int i = 0;
#pragma omp parallel num_threads(2)
{
    a++;
    #pragma omp barrier
    #pragma omp single
    {
          cout<<a<<endl;
    }
}
```

This version of the code is entirely correct: the program will always print the "2" string. Please note that this version does not contain the flush directive since it is implicitly included in the barrier directive.

Now, let us consider one more example of missing synchronization. The example is taken from the MSDN Library [8].

Incorrectly:

```
struct MyType
{
    ~MyType();
};

MyType threaded_var;
#pragma omp threadprivate(threaded_var)
int main()
{
    #pragma omp parallel
    {
      ...
    }
}
```

The code is incorrect, because there is no synchronization at exit from the parallel section. As the result, when the application's process execution finishes, some of the threads will still exist and they will not get a notification about the fact that the process execution is finished. The destructor of the threaded_var variable will actually be called only in the main thread. Since the variable is threadprivate, its copies created in other threads will not be destroyed and a memory leak will occur. It is necessary to implement synchronization manually in order to avoid this problem.

Correctly:

```
struct MyType
{
    ~MyType();
};

MyType threaded_var;
#pragma omp threadprivate(threaded_var)
int main()
{
    #pragma omp parallel
    {
           ...
           #pragma omp barrier
    }
}
```

18. An external variable is specified as threadprivate not in all units

We're beginning to discuss the most troublesome errors: the errors related to the OpenMP memory model. And this one is the first error of this type. The concurrent access to shared memory can also be treated as an error related to the OpenMP memory model since the error is related to shared variables and all global-scope variables are shared by default in OpenMP.

Before we start discussing memory model errors, please note that they all are related to private, firstprivate, lastprivate and threadprivate variables. One can avoid most of the errors if he avoids using the threadprivate directive and the private clause. We recommend declaring the needed variables as local variables in parallel sections instead.

Now, when you're warned, let's start discussing the memory model errors. We'll start with the threadprivate directive. The directive is usually applied to global variables, including external variables declared in another units. In this case the directive should be applied to the variable in all the units in which the variable is used. This rule is described in the abovementioned MSDN Library article [8].

A special case of this rule is another rule described in the same article: the threadprivate directive cannot be applied to variables declared in a DLL which will be loaded via the LoadLibrary function or the /DELAYLOAD linker option (since the LoadLibrary function is used implicitly in this case).

19. Uninitialized local variables

When a thread starts, local copies of threadprivate, private and lastprivate variables are created for this thread. The copies are unitialized by default. Therefore, any attempt to work with the variables without initializing them, will cause a run-time error.

Incorrectly:

```
int a = 0;
#pragma omp parallel private(a)
{
    a++;
}
```

Correctly:

```
int a = 0;
#pragma omp parallel private(a)
{
    a = 0;
    a++;
}
```

Please note that there is no need to use synchronization and the flush directive since every thread has its own copy of the variable.

20. Forgotten threadprivate directive

Since the threadprivate directive is applied only once and used for global variables declared in the beginning of a unit, it's easy to forget about the directive: for example, when it's necessary to modify a unit created half a year ago. As the result, the developer will expect a global variable to become shared, as it should be by default. However, the variable will become local for every parallel thread. According to the OpenMP specification [3], the variable's value after a parallel section is unpredictable in this case.

Incorrectly:

```
int a;
#pragma omp threadprivate(a)

int _tmain(int argc, _TCHAR* argv[])
{
    ...
    a = 0;
    #pragma omp parallel
    {
          #pragma omp sections
          {
                 #pragma omp section
                 {
                       a += 3;
                 }
                 #pragma omp section
                 {
                       a += 3;
                 }
           }
           #pragma omp barrier
    }
    cout << "a = " << a << endl;
}
```

The program will behave as described in the specification: sometimes "6" (the value the developer expects) will be printed in a console window. Sometimes, however, the program will print "0". This result is more logical, since 0 is the value assigned to the variable before the parallel section. In theory, the same behavior should be observed if the a variable is declared as private or firstprivate. In practice, however, we have reproduced the behavior only with the threadprivate directive. Therefore, the example above contains this directive. In addition, this case is the most probable.

This fact, however, does not mean that the behavior in the other two cases will be correct in all other implementations; so, one should consider the cases too.

Unfortunately, it is difficult to provide a good solution in this case, because removing the threadprivate directive will change the program's behavior and declaring a threadprivate variable as shared is forbidden by OpenMP syntax rules. The only possible workaround is to use another variable.

Correctly:

```
int a;
#pragma omp threadprivate(a)

int _tmain(int argc, _TCHAR* argv[])
{
    ...
    a = 0;
    int b = a;
    #pragma omp parallel
    {
          #pragma omp sections
          {
                 #pragma omp section
                 {
                       b += 3;
                 }
                 #pragma omp section
                 {
                       b += 3;
                 }
           }
           #pragma omp barrier
    }
    a = b;
    cout << "a = " << a << endl;
}
```

In this version the a variable becomes a shared variable for the parallel section. Of course, this solution is not the best one. However, this solution guarantees that the old code will not change its behavior.

We recommend that beginners use the default(none) clause to avoid such problems. The clause will make the developer specify access modes for all global variables used in a parallel section. Of course, this will make your code grow, but you will avoid many errors and the code will become more readable.

21. Forgotten private clause

Let's consider a scenario similar to the previous case: a developer needs to modify a unit created some time ago and the clause defining a variable's access mode is located far enough from the code fragment to be modified.

Incorrectly:

```
int a;
#pragma omp parallel private(a)
{
...
a = 0;
#pragma omp for
for (int i = 0; i < 10; i++)
{
    #pragma omp atomic
    a++;
}
#pragma omp critical
{
   cout << "a = " << a;
}
}
```

This error seems to be an equivalent of the previous one. However, it is not true. In the previous case the result was printed after a parallel section and in this case the value is printed from a parallel section. As the result, if the variable's value before the loop is equal to zero, the code will print "5" instead of "10" on a two-processor machine. The cause of the behavior is that the work is shared between two threads. Each thread will get its own local copy of the a variable and increase the variable five times instead of the expected ten times. Moreover, the resulting value will depend on the number of threads executing the parallel section. By the way, the error will also occur if one uses the firstprivate clause instead of the private clause.

Possible solutions are similar to the ones provided for the previous case: one should either significantly modify all older code or modify the new code so that it will be compatible with the behavior of the old code. In this case the second solution is more elegant than the one provided for the previous case.

Correctly:

```
int a;
#pragma omp parallel private(a)
{
...
a = 0;
#pragma omp parallel for
for (int i = 0; i < 10; i++)
{
    #pragma omp atomic
a++;
}
#pragma omp critical
{
    cout << "a = " << a;
}
}
```

22. Incorrect worksharing with private variables

The error is similar to the previous one and opposite to the "Unnecessary parallelization" error. In this case, however, the error can be caused by another scenario.

Incorrectly:

```
int a;
#pragma omp parallel private(a)
{
    a = 0;
    #pragma omp barrier
    #pragma omp sections
    {
          #pragma omp section
          {
                 #pragma omp atomic
                 a+=100;
           }
           #pragma omp section
           {
                 #pragma omp atomic
                 a+=1;
           }
    }
    #pragma omp critical
{
    cout << "a = " << a << endl;
}
}
```

In this case a developer wanted to increase the value of each local copy of the a variable by 101 and used the sections directive for this purpose. However, since the parallel keyword was not specified in the directive, no additional parallelization was made. The work was shared between the same threads. As the result, on a two-processor machine one thread will print "1" and the other one will print "100". If the number of threads is increased, the results will be even more unexpected. By the way, if the a variable is not declared as private, the code will become correct.

In the sample above it is necessary to perform additional code parallelization.

Correctly:

```
int a;
#pragma omp parallel private(a)
{
    a = 0;
    #pragma omp barrier
    #pragma omp parallel sections
    {
           #pragma omp section
           {
                 #pragma omp atomic
                 a+=100;
           }
           #pragma omp section
           {
                 #pragma omp atomic
                 a+=1;
          }
    }
    #pragma omp critical
{
    cout<<"a = "<<a<<endl;
}
}
```

23. Careless usage of the lastprivate clause

OpenMP specification states that the value of a lastprivate variable from the sequentially last iteration of the associated loop, or the lexically last section directive is assigned to the variable's original object. If no value is assigned to the lastprivate variable during the corresponding parallel section, the original variable has indeterminate value after the parallel section. Let's consider an example similar to the previous one.

Incorrectly:

```
int a = 1;
#pragma omp parallel
{
    #pragma omp sections lastprivate(a)
    {
           #pragma omp section
           {
                 ...
                 a = 10;
          }
          #pragma omp section
          {
                 ...
          }
    }
#pragma omp barrier
}
```

This code may potentially cause an error. We were unable to reproduce this in practice; however, it does not mean that the error will never occur.

If a developer really needs to use the lastprivate clause, he should know exactly what value would be assigned to the variable after a parallel section. In general case an error may occur if an unexpected value is assigned to the variable. For example, the developer may expect that the variable will get a value from the thread that finishes its execution last, but the variable will get a value from a lexically last thread. To solve this problem the developer should simply swap the sections' code.

Correctly:

```
int a = 1;
#pragma omp parallel
{
    #pragma omp sections lastprivate(a)
    {
           #pragma omp section
           {
                 ...
           }
           #pragma omp section
           {
                 ...
                 a = 10;
           }
    }
#pragma omp barrier
}
```

24. Unexpected values of threadprivate variables in the beginning of parallel sections

This problem is described in the OpenMP specification [3]. If the value of a threadprivate variable is changed before a parallel section, the value of the variable in the beginning of the parallel section is indeterminate.

Unfortunately, the sample code provided in the specification, cannot be compiled in Visual Studio since the compiler does not support dynamic initialization of threadprivate variables. Therefore, we provide another, less complicated, example.

Incorrectly:

```
int a = 5;
#pragma omp threadprivate(a)

int _tmain(int argc, _TCHAR* argv[])
{
...
a = 10;
#pragma omp parallel num_threads(2)
{
    #pragma omp critical
    {
          printf("\nThread #%d: a = %d", omp_get_thread_num(),a);
    }
}
getchar();
return 0;
}
```

After the program execution one of the threads will print "5" and the other will print "10". If the a variable initialization is removed, the first thread will print "0" and the second one will print "10". One can get rid of the unexpected behavior only by removing the second assignment. In this case both threads will print "5" (in case the initialization code is not removed). Of course, such modifications will change the code's behavior. We describe them only to show OpenMP behavior in the two cases.

The resume is simple: never rely upon on your compiler when you need to initialize a local variable. For private and lastprivate variables an attempt to use uninitialized variables will cause a run-time error, which has already been described above. The error is at least easy to localize. The threadprivate directive, as you can see, may lead to unexpected results without any errors or warnings. We strongly recommend that you do not use this directive. In this case your code will become much more readable and the code's behavior will be easier to predict.

Correctly:

```
int a = 5;

int _tmain(int argc, _TCHAR* argv[])
{
...
a = 10;
#pragma omp parallel num_threads(2)
{
    int a = 10;
    #pragma omp barrier
    #pragma omp critical
    {
          printf("\nThread #%d: a = %d", omp_get_thread_num(),a);
    }
}
getchar();
return 0;
}
```

25. Some restrictions of private variables

The OpenMP specification provides multiple restrictions concerning private variables. Some of the restrictions are automatically checked by the compiler. Here is the list of restrictions which are not checked by the compiler:

- A private variable must not have a reference type.
- If a lastprivate variable is an instance of a class, the class should have a copy constructor defined.
- A firstprivate variable must not have a reference type.
- If a firstprivate variable is an instance of a class, the class should have a copy constructor defined.
- A threadprivate variable must not have a reference type.

In fact, all the restrictions result into two general rules: 1) a private variable must not have a reference type 2) if the variable is an instance of a class, the class should have a copy constructor defined. The causes of the restrictions are obvious.

If a private variable has a reference type, each thread will get a copy of this reference. As the result, both threads will work with shared memory via the reference.

The restriction, concerning the copy constructor, is quite obvious too: if a class contains a field which has a reference type, it will be impossible to copy an instance of this class memberwise correctly. As the result, both threads will work with shared memory, just like in the previous case.

An example demonstrating the problems is too large and is unlikely necessary. One should only remember a single common rule. If it is necessary to create a local copy of an object, an array or a memory fragment addressed via a pointer, the pointer should remain a shared variable. Declaring the variable as private is meaningless. The referenced data should be either copied explicitly or (when you're dealing with objects) entrusted to the compiler which uses the copy constructor.

26. Private variables are not marked as such

The error is described in the article [1]. The cause of the problem is that a variable which is supposed to be private was not marked as such and is used as a shared variable since this access mode is applied to all variables by default.

We recommend that you use the default(none) clause, which has already been mentioned above, to diagnose the error.

As you can see, the error is rather abstract and it is difficult to provide an example. However, the article [9] describes a situation in which the error occurs quite explicitly.

Incorrectly:

```
int _tmain(int argc, _TCHAR* argv[])
{
 const size_t arraySize = 100000;
 struct T {
   int a;
   size_t b;
 };
 T array[arraySize];
 {
   size_t i;
   #pragma omp parallel sections num_threads(2)
   {
     #pragma omp section
     {
       for (i = 0; i != arraySize; ++i)
         array[i].a = 1;
     }
     #pragma omp section
     {
       for (i = 0; i != arraySize; ++i)
         array[i].b = 2;
     }
   }
 }

 size_t i;
 for (i = 0; i != arraySize; ++i)
 {
   if (array[i].a != 1 || array[i].b != 2)
   {
     _tprintf(_T("OpenMP Error!\n"));
     break;
   }
 }
 if (i == arraySize)
```

```
   _tprintf(_T("OK!\n"));
    getchar();
    return 0;
}
```

The program's purpose is simple: an array of two-field structures is initialized from two threads. One thread assigns 1 to one of the fields and the other assigns 2 to the other field. After this operation the program checks whether the array was initialized successfully.

The cause of the error is that both threads use a shared loop variable. In some cases the program will print the "OpenMP Error!" string. In other cases an access violation will occur. And only in rare cases the "OK!" string will be printed. The problem can be easily solved by declaring the loop variable as local.

Correctly:

```
...
   #pragma omp parallel sections num_threads(2)
   {
     #pragma omp section
     {
       for (size_t i = 0; i != arraySize; ++i)
         array[i].a = 1;
     }
     #pragma omp section
     {
       for (size_t i = 0; i != arraySize; ++i)
         array[i].b = 2;
     }
   }
 }
...
```

The article [1] contains a similar example, concerning for loops (the example is considered as a separate error). The author states that loop variable of a for loop shared via the for OpenMP directive should be declared as local. The situation seems to be equal to the one described above at fist sight. However, it not true.

According to the OpenMP standard loop variables are converted to private implicitly in such cases, even if the variable is declared as shared. The compiler will report no warnings after performing this conversion. This is the case described in the article [1], and the conversion is performed in this case indeed. However, in our example the loop is shared between threads using the sections directive, not the for directive, and in this case the conversion is not performed.

The resume is quite obvious: loop variables must never be shared in parallel sections. Even if the loop is shared between threads via the for directive, you should not rely on implicit conversion in this case.

27. Parallel array processing without iteration ordering

Parallelized for loops execution was not ordered in all previous examples (except the one concerning the ordered directive syntax). The loops were not ordered because there was no need to do this. In some cases, however, the ordered directive is necessary. In particular, you need to use the directive if an iteration result depends on a previous iteration result (this situation is described in the article [6]). Let's consider an example.

Incorrectly:

```
int* arr = new int[10];
for(int i = 0; i < 10; i++)
    arr[i] = i;
#pragma omp parallel for
for (int i = 1; i < 10; i++)
    arr[i] = arr[i - 1];
for(int i = 0; i < 10; i++)
    printf("\narr[%d] = %d", i, arr[i]);
```

In theory the program should have printed a sequence of zeros. However, on a two-processor machine the program will print a number of zeros along with a number of fives. This behavior is caused by the fact that iterations are usually split equally between the threads by default. The problem can be easily solved using the ordered directive.

Correctly:

```
int* arr = new int[10];
for(int i = 0; i < 10; i++)
    arr[i] = i;
#pragma omp parallel for ordered
for (int i = 1; i < 10; i++)
{
    #pragma omp ordered
    arr[i] = arr[i - 1];
}
for(int i = 0; i < 10; i++)
    printf("\narr[%d] = %d", i, arr[i]);
```

Performance errors

1. Unnecessary flush directive

All errors considered above affected the analyzed programs' logic and were critical. Now, let us consider errors which only affect a program's performance without affecting the program's logic. The errors are described in the article [1].As we have already mentioned above, the flush directive is often implied. Therefore, explicit flush directive in these cases is unnecessary. Unnecessary flush directive, especially the one used without parameters (in this case all shared memory is synchronized), can significantly slow down a program's execution.Here are the cases in which the directive is implied and there is no need to use it:The barrier directive

- At entry to and at exit from critical
- At entry to and at exit from ordered
- At entry to and at exit from parallel
- At exit from for
- At exit from sections
- At exit from single
- At entry to and at exit from parallel for
- At entry to and at exit from parallel sections

2. Using critical sections or locks instead of the atomic directive

The atomic directive works faster than critical sections, since many atomic operations can be replaced with processor commands. Therefore, it is more preferable to apply this directive when you need to protect shared memory during elementary operations. According to the OpenMP specification, the directive can be applied to the following operations:x binop= exprx++++xx----xHere x is a scalar variable, expr is a scalar statement which does not involve the x variable, binop is +, *, -, /, &, ^, |, <<, or >> operator which was not overloaded. In all other cases the atomic directive cannot be used (this condition is checked by the compiler).

Here is a list of shared memory protection means sorted by performance in descending order: atomic, critical, omp_set_lock.

3. Unnecessary concurrent memory writing protection

Any protection slows down the program's execution and it does not matter whether you use atomic operations, critical sections or locks. Therefore, you should not use memory protection when it is not necessary.

A variable should not be protected from concurrent writing in the following cases:

- If a variable is local for a thread (also, if the variable is threadprivate, firstprivate, private or lastprivate).
- If the variable is accessed in a code fragment which is guaranteed to be executed by a single thread only (in a master or single section).

4. Too much work in a critical section

Critical sections always slow down a program's execution. Firstly, threads have to wait for each other because of critical sections, and this decreases the performance increase you gain using code parallelization. Secondly, entering and leaving a critical section takes some time.

Therefore, you should not use critical sections where it is not necessary. We do not recommend that you place complex functions' calls into critical sections. Also, we do not recommend putting code which does not work with shared variables, objects or resources in critical sections. It is rather difficult to give exact recommendations on how to avoid the error. A developer should decide whether a code fragment should be put into critical section in every particular case.

5. Too many entries to critical sections

As we have already mentioned in the previous error description, entering and leaving a critical section takes some time. Therefore, if the operations are performed too often, this may decrease a program's performance. We recommend that you decrease the number of entries to critical sections as much as possible. Let's consider a slightly modified example from the article [1].

Incorrectly:

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i )
{
    #pragma omp critical
    {
          if (arr[i] > max) max = arr[i];
    }
}
```

If the comparison is performed before the critical section, the critical section will not be entered during all iterations of the loop.

Correctly:

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i )
{
    #pragma omp flush(max)
    if (arr[i] > max)
    {
          #pragma omp critical
          {
                if (arr[i] > max) max = arr[i];
          }
    }
}
```

Such a simple correction may allow you to increase your code's performance significantly and you should not disregard this advice.

Conclusion

This paper provides the most complete list of possible OpenMP errors, at least at the moment the paper was written. The data provided in this article were collected from various sources as long as from authors' practice. Please note that all the errors are not diagnosed by standard compilers. Now, let us provide a short description of all the errors with the corresponding conclusions.

Error	Conclusion
1. Missing /openmp compiler option	You should enable the option at the moment you create your project.
2. Missing parallel keyword	You should be accurate about the syntax of the directives you use.
3. Missing omp keyword	You should be accurate about the syntax of the directives you use.
4. Missing for keyword	You should be accurate about the syntax of the directives you use.
5. Unnecessary parallelization	You should be accurate about the syntax of the directives you use and understand their meaning.
6. Incorrect usage of the ordered clause	It is necessary to watch over the syntax of the directives you use.
7. Redefining the number of threads in a parallel section	The number of threads cannot be changed in a parallel section.
8. Using a lock variable without initializing the variable	A lock variable must be initialized via the omp_init_lock function call.
9. Unsetting a lock from another thread	If a thread uses locks, both the lock (omp_set_lock, omp_test_lock) and unlock (omp_unset_lock) functions must be called by this thread.
10. Using a lock as a barrier	If a thread uses locks, both the lock (omp_set_lock, omp_test_lock) and unlock (omp_unset_lock) functions must be called by this thread.
11. Threads number	Your code's behavior must not depend on the number of threads which

dependency	execute the code.
12. Incorrect usage of dynamic threads creation	If you really need to make your code's behavior depend on the number of threads, you must make sure that the code will be executed by the needed number of threads (dynamic threads creation must be disabled). We do not recommend using dynamic threads creation.
13. Concurrent usage of a shared resource	Concurrent shared resource access must be protected by a critical section or a lock.
14. Shared memory access unprotected	Concurrent shared memory access must be protected as an atomic operation (the most preferable option), critical section or a lock.
15. Using the flush directive with a reference type	Applying the flush directive to a pointer is meaningless since only the variable's value (a memory address, not the addressed memory) is synchronized in this case.
16. Missing flush directive	Missing flush directive may cause incorrect memory read/write operations.
17. Missing synchronization	Missing synchronization may also cause incorrect memory read/write operations.
18. An external variable is specified as threadprivate not in all units	If a threadprivate variable is an external variable, it must be declared as threadprivate in all the units, which use the variable. We recommend that you do not use the threadprivate directive and the private, firstprivate, lastprivate clauses. We recommend that you declare local variables in parallel sections and perform first/last assignment operations (if they are necessary) with a shared variable.
19. Uninitialized private variables	All private and lastprivate variables are uninitialized by default. You cannot use the variables until you initialize them. We recommend that you do not use the threadprivate directive and the private, firstprivate, lastprivate clauses. We recommend that you declare local variables in parallel sections and perform first/last assignment operations (if they are necessary) with a shared variable.
20. Forgotten threadprivate directive	A forgotten threadprivate directive may affect an entire unit's behavior. We recommend that you do not use the threadprivate directive and the private, firstprivate, lastprivate clauses. We recommend that you declare local variables in parallel sections and perform first/last assignment operations (if they are necessary) with a shared variable.
21. Forgotten private clause	You must control access modes of your variables. We recommend that developers who are new to OpenMP use the default(none) clause so that they will have to specify access modes explicitly. We recommend that you do not use the threadprivate directive and the private, firstprivate, lastprivate clauses. We recommend that you declare local variables in parallel sections and perform first/last assignment operations (if they are necessary) with a shared variable.
22. Incorrect worksharing with private variables	If you parallelize a code fragment which works with private variables using the threads in which the variables were created different threads will get different values of the variables.
23. Careless usage of the	If you are using the lastprivate clause, you must know exactly what value will be assigned to the variable after the parallel section. We recommend that you do not use the threadprivate directive and the private, firstprivate,

lastprivate clause	lastprivate clauses. We recommend that you declare local variables in parallel sections and perform first/last assignment operations (if they are necessary) with a shared variable.
24. Unexpected values of threadprivate variables in the beginning of parallel sections	A threadprivate variable's value is unpredictable in the beginning of a parallel section, especially if a value was assigned to the variable before the parallel section. We recommend that you do not use the threadprivate directive and the private, firstprivate, lastprivate clauses. We recommend that you declare local variables in parallel sections and perform first/last assignment operations (if they are necessary) with a shared variable.
25. Some restrictions of private variables	Private variables must not have reference type, since it will cause concurrent shared memory access. Although the variables will be private, the variables will still address the same memory fragment. Class instances declared as private must have explicit copy constructor, since an instance containing references will be copied incorrectly otherwise.
26. Private variables are not marked as such	You must control access modes of your variables. We recommend that developers who are new to OpenMP use the default(none) clause so that they will have to specify access modes explicitly. In particular, loop variables must always be declared as private or local variables.
27. Parallel array processing without iteration ordering	If an iteration execution depends on the result of a previous iteration, you must use the ordered directive to enable iterations ordering.
1. Unnecessary flush directive	There is no need to use the flush directive in the cases when the directive is implied.
2. Using critical sections or locks instead of the atomic directive	We recommend that you use the atomic directive to protect elementary operations when it is possible, since using locks or critical sections slows down you program's execution.
3. Unnecessary concurrent memory writing protection	There is no need protect private or local variables. Also, there is no need to protect a code fragment which is executed by a single thread only.
4. Too much work in a critical section	Critical sections should contain as little work as possible. You should not put a code fragment which does not work with shared memory into a critical section. Also we do not recommend putting a complex function calls into a critical section.
5. Too many entries to critical sections	We recommend that you decrease the number of entries to and exits from critical sections. For example, if a critical section contains a conditional statement, you can place the statement before the critical section so that the critical section is entered only if the condition is true.

Table 1 - A short list of OpenMP errors.

All the errors can be divided into three general categories:

- Ignorance of the OpenMP syntax.
- Misunderstanding of the OpenMP principles.
- Incorrect memory processing (unprotected shared memory access, lack of synchronization, incorrect variables' access mode, etc.).

Of course, the errors list provided in this paper is not complete. There are many other errors which were not considered here. It is possible that more complete lists will be provided in new articles on this topic.

Most of the errors can be diagnosed automatically by a static analyzer. Some (only a few) of them can be detected by Intel Thread Checker. Also, some errors are detected by compilers other than the one used in Visual Studio. However, a specialized tool for detecting such errors has not been created yet. In particular, Intel Thread Checker detects concurrent shared memory access, incorrect usage of the ordered directive and missing for keyword in the #pragma omp parallel for directive [1].

A program for visual representation of code parallelization and access modes could also be useful for developers and has not been created yet.

The authors start working on the VivaMP static analyzer at the moment. The analyzer will diagnose the errors listed above and, maybe, some other errors. The analyzer will significantly simplify errors detection in parallel programs (note that almost all such errors cannot be stably reproduced). Additional information on the VivaMP project can be found on the project page: (http://www.viva64.com/vivamp.php).

References

1. Michael Suess, Claudia Leopold, Common Mistakes in OpenMP and How To Avoid Them - A Collection of Best Practices, http://www.viva64.com/go.php?url=100
2. OpenMP Quick Reference Sheet, http://www.viva64.com/go.php?url=101
3. OpenMP C and C++ Application Program Interface specification, version 2.0, http://www.viva64.com/go.php?url=102
4. Yuan Lin, Common Mistakes in Using OpenMP 1: Incorrect Directive Format, http://www.viva64.com/go.php?url=103
5. Richard Gerber, Advanced OpenMP Programming, http://www.viva64.com/go.php?url=104
6. Kang Su Gatlin and Pete Isensee. Reap the Benefits of Multithreading without All the Work, http://www.viva64.com/go.php?url=105
7. Yuan Lin, Common Mistakes in Using OpenMP 5: Assuming Non-existing Synchronization Before Entering Worksharing Construct, http://www.viva64.com/go.php?url=106
8. MSDN Library article on 'threadprivate' OpenMP directive, http://www.viva64.com/go.php?url=107
9. Andrey Karpov, Evgeniy Ryzhkov, Adaptation of the technology of the static code analyzer for developing parallel programs, http://www.viva64.com/art-3-2-486346462.html